Étape 3 : Environnement et nourriture

Jamila Sam & Jean-Cédric Chappelier, 2018

Version : 1.1

Buts

- Mettre en place le lieu de vie des futurs animaux et leur fournir de la nourriture.
- Comprendre la mise en pratique de l'encapsulation (orientée-objet) au travers de « *vues* ».

Description

La faune que nous allons simuler va évoluer dans un environnement. Bien modéliser un tel environnement, en respectant les principes fondamentaux de l'orienté-objet, est important. Cela permettra notamment de limiter au maximum les dépendances entre les différentes composantes de notre simulation. Les dépendances inutiles entre composantes d'un programme vont en effet à l'encontre du principe de « séparation des soucis » et peuvent nuire à une bonne encapsulation, comme nous en discuterons un peu plus loin.

L'environnement est responsable de répertorier toutes les entités nécessaires à la vie des animaux comme les sources de nourritures ou, plus tard, les fourmilières, ou encore la phéromone déposée sur le sol.

Dans cette étape-ci, nous commençons par mettre en place de la nourriture, dans l'environnement bien sûr ! La nourriture sera placée au hasard dans l'environnement au moyen d'une entité basique, un générateur de nourriture, qu'il s'agira de modéliser aussi.

Ceci constitue notre première interaction entité-environnement (l'entité étant ici un générateur de nourriture). L'environnement sera bien sûr complété au fur et à mesure des différentes étapes. La nourriture, quant à elle, sera plus tard collectée par les fourmis.

Minimiser les dépendances entre la nourriture et l'environnement qui la contient va nous permettre de réviser une problématique fondamentale en programmation

orientée-objet : l'« encapsulation », que nous allons réaliser au travers de différentes interfaces, différentes « *vues* » sur l'environnement.

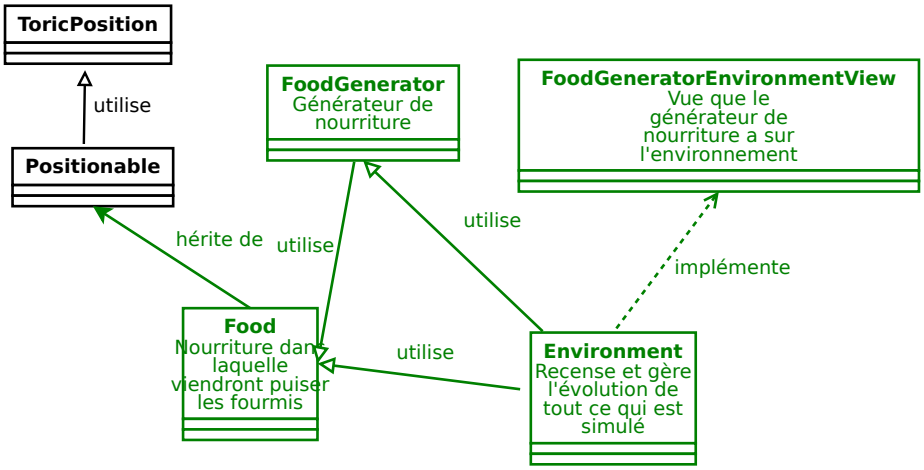

Figure 1: Architecture codée au terme de l'étape 3 (en noir les classes déjà codées, en vert les nouvelles classes)

La figure 1 donne une vue d'ensemble de l'architecture à laquelle vous allez aboutir au terme de cette étape.

Implémentation

Classe `Food`

Commençons simplement par la nourriture. La classe `Food` est la classe, finale, modélisant la nourriture consommable par les fourmis. Elle est caractérisée par une position torique (vous considérerez qu'une `Food` est un `Positionable`) ainsi qu'une valeur, de type `double`, indiquant la quantité de nourriture disponible à cette position là.

Vous doterez la classe `Food` :

- d'un constructeur initialisant la position torique et la quantité au moyen de valeurs passées en arguments ; si la valeur passée en argument pour la quantité est négative, il faut alors initialiser la quantité de nourriture à 0.0 ;
- d'une méthode `getQuantity()` permettant de connaître la quantité de nourriture disponible ;
- d'une méthode `takeQuantity(double)` permettant de prélever (c.-à-d. soustraire) une quantité donnée de nourriture ;

cette méthode retournera la quantité qui a *pu* être prélevée (on ne peut pas prélever plus que ce qui est disponible) ; si l'argument est négatif, il faudra lancer une `IllegalArgumentException` ; faites attention à ce que la quantité de nourriture restante ne soit pas négative !

- d'une méthode `toString()` retournant une représentation textuelle d'une `Food` selon le format de l'exemple suivant :

```
Position : 1.2, 2.3
Quantity : 4.70
```

Les libellés `Position` et `Quantity` sont importants pour une bonne intégration avec l'interface graphique à venir. Vous veillerez à réutiliser proprement les méthodes `toString` héritées de plus haut.

Indication : Pour formatter la quantité avec deux chiffres après la virgule, vous pourrez recourir à `String.format`, selon l'exemple suivant :

```
String.format("Quantity : %.2f", getQuantity())
```

Classe `Environment` – introduction

Cadre général

La classe `Environment` décrit le monde dans lequel les animaux évoluent. En ce sens, elle comportera différentes collections d'entités simulées (une collection d'animaux, une de nourritures, une de phéromones, etc.). Elle comportera aussi un générateur (classe `FoodGenerator` expliqué plus en détails ci-dessous) qui aura pour vocation de placer aléatoirement des sources de nourriture dans l'environnement.

L'environnement est responsable de la bonne évolution au cours du temps de toutes les entités qu'il contient. A ce titre, il doit appeler à chaque intervalle de temps de simulation, les méthodes de « mise-à-jour » des toutes ces entités. Par exemple, la « mise-à-jour » du générateur est ce qui va lui permettre de décider qu'il est temps de créer une nouvelle nourriture placée aléatoirement. Les méthodes de « mise-à-jour » seront évidemment spécifiques aux entités (une fourmi soldate n'évolue pas comme un générateur de nourriture !) et donc typiquement polymorphiques.

Si nous nous intéressons à l'interaction basique entre le générateur de nourritures et son environnement, nous pouvons voir qu'elle pose déjà une problématique de conception intéressante : l'environnement a un impact sur le générateur puisqu'il lui demande de se mettre à jour et le générateur a un impact sur l'environnement puisque qu'il va placer de nouveaux éléments dans sa collection de nourritures. Il y a donc une dépendance, dans les deux sens, entre les classes `Environment` et `FoodGenerator`. Cette même problématique va se poser pour

les autres entités (l'environnement demande au fourmis d'évoluer et les fourmis agissent sur l'environnement en y déposant de la phéromone par exemple).

La première question qui se pose alors est comment se traduit ce type de dépendance dans la conception et donc très concrètement, comment une entité a t-elle connaissance de l'environnement sur lequel elle peut avoit un impact ?

Une façon (simple) de procéder est de faire en sorte que la méthode de « mise-à-jour » des entités prenne en paramètre l'environnement dans lequel elles évoluent. Cette approche directe pose néanmoins quelques problèmes...

Augmenter l'encapsulation au travers de « vues »

Reprenons l'exemple du générateur de nourritures. Pour qu'il puisse impacter l'environnement, la classe `Environment` devra lui donner accès à une méthode permettant d'ajouter un nouvel élément à sa collection de nourritures. L'accès à cette méthode devrait cependant être restreint au générateur ; aucune autre entité ne devrait pouvoir ajouter de la nourriture à l'environnement. Par exemple, les fourmilières qui peuvent ajouter des fourmis à l'environnement ne devraient pas pouvoir appeler cette méthode (et réciproquement une source de nourriture ne doit pas pouvoir appeler la méthode de l'environnement qui lui permet de rajouter des fourmis !).

Avec l'approche mentionnée ci-dessus, n'importe quelle entité, du fait qu'elle accède à l'environnement par le biais du paramètre de sa méthode de mise-à-jour, peut sans restriction appeler des méthodes qui ne lui sont pas destinées (la méthode de mise-à-jour d'une fourmilière peut tout à fait ajouter à l'environnement de la nourriture comme bon lui semble !). Ceci nuit clairement à une bonne encapsulation.

Une solution à ce problème est d'introduire la notion de « vue » sur l'environnement : chaque type d'entité interagissant avec l'environnement aura une vue limitée sur celui-ci, spécifique à ce type d'entité. Cela signifie qu'un type d'entité n'aura pas forcément accès aux mêmes informations qu'un autre type d'entité. Cela permet aussi de contrôler les actions que pourront entreprendre les différentes entités. Par exemple, un générateur de nourriture verra les positions où il peut placer de la nourriture, mais ne verra pas les fourmilières ni les phéromones et, surtout, ne pourra pas placer de fourmi ni faire évaporer de la phéromone. De même, il sera possible pour une fourmilière d'ajouter des fourmis dans l'environnement, mais une termite ne pourra pas le faire car sa vue de l'environnement ne le lui permet pas.

Les méthodes de mise-à-jour des entités ne prendront alors plus en paramètre une instance d'`Environment`, mais une instance de la vue qu'elles ont sur l'environnement. Concrètement, on peut traduire cette contrainte en utilisant les interfaces Java : chaque type d'entité sera associé à une interface qui lui est propre et qui correspond à la vue qu'elle a sur l'environnement. La classe

`Environment` implémentera toutes ces interfaces. Lors de l'appel des méthodes de mise-à-jour des entités, une instance `Environment` sera bien passée en paramètre, mais les entités ne la percevront plus comme telle ! Elles ne la percevront plus que comme la vue limitée qu'elles doivent en avoir.

Cette couche d'abstraction supplémentaire illustre la puissance des interfaces comme outil d'encapsulation. Elle permet à la classe `Environment` de contrôler les actions des entités qui la « voient ».

Concrètement

Concrètement, à la classe `FoodGenerator` correspondra une interface `FoodGeneratorEnvironmentView` qui modélisera l'interaction qu'aura un générateur de nourriture avec l'environnement (ajout de nourriture).

Puis nous allons faire en sorte que la classe `Environment` implémente l'interface `FoodGeneratorEnvironmentView` et redéfinisse la méthode adéquate permettant d'ajouter de la nourriture.

Nous ajouterons à la classe `Environment` une méthode publique `void update(Time dt)` qui aura la responsabilité d'appeler toutes les méthodes de « mise-à-jour » des entités ; pour le moment uniquement celle de ses `FoodGenerator`. Enfin, le `FoodGenerator` prendra un `FoodGeneratorEnvironmentView` comme paramètre de sa méthode de « mise-à-jour ».

Allons y !

Classe `FoodGeneratorEnvironmentView`

Commençons par le composant le plus simple (une interface en fait) : la vue qu'ont les générateurs de nourriture de l'environnement. Comme `FoodGenerator` a besoin d'interagir avec l'environnement de façon spécifique, nous allons doter sa vue `FoodGeneratorEnvironmentView` d'une méthode dédiée à cela :

```
void addFood(Food food)
```

Etant donné que `FoodGeneratorEnvironmentView` est une interface, et qu'elle ne peut de ce fait accéder à la collection de nourritures comme attribut, la définition de cette méthode n'y est pas donnée, mais se trouvera dans `Environment`.

Et voilà ! C'est tout pour `FoodGeneratorEnvironmentView`.

Classe `FoodGenerator`

La classe `FoodGenerator` a pour but de générer périodiquement, à des espaces de temps réguliers, de la nourriture dans l'environnement. Cette classe sera

finale.

Pour la classe `FoodGenerator` elle-même : créez la et dotez la d'un attribut de type `Time` (cette classe vous est fournie dans `utils`). Cet attribut, utilisé comme compteur, permettra de mesurer le temps écoulé depuis la précédente génération de nourriture. Le constructeur par défaut initialisera ce temps à `Time.ZERO`.

Pour faire évoluer ce compteur au cours du temps, il faut que la classe `FoodGenerator` dispose d'une méthode

```
void update(FoodGeneratorEnvironmentView env, Time dt)
```

gérant l'évolution de ses instances à chaque écoulement de pas de temps `dt`.

Cette méthode `update()` met en œuvre l'algorithme suivant :

1. augmenter le compteur interne de `dt` ;
2. tant que sa valeur dépasse le seuil spécifié par le paramètre `FOOD_GENERATOR_DELAY` (valeur donnée par l'appel `Context.getConfig().getTime(Config.FOOD_GENER` :
    1. décrémenter ce compteur interne de cette même valeur ;
    2. ajouter à l'environnement une quantité de nourriture placée aléatoirement comme expliqué ici :
        - la quantité à placer est déterminée par appel à la méthode statique

          `getValue(double min, double max)`

        de la classe `ch.epfl.moocprog.random.UniformDistribution` avec les valeurs d'appel `getConfig().getDouble(NEW_FOOD_QUANTITY_MIN)` et `getConfig().getDouble(NEW_FOOD_QUANTITY_MAX)` ; cela tire une valeur au hasard uniformément entre ces deux bornes ;
        - les coordonnées de la position où placer la nourriture sont chacune tirée au hasard en utilisant un appel à la méthode statique

          `getValue(double mu, double sigma2)`

        de la classe `ch.epfl.moocprog.random.NormalDistribution` avec comme valeur pour `mu` `taille / 2.0` et pour `sigma2` `taille * taille / 16.0`, où `taille` vaut `Context.getConfig().getInt(WORLD_WIDTH)` pour l'abscisse `x` et vaut `Context.getConfig().getInt(WORLD_HEIGTH)` pour l'ordonnée `y`.

Remarques :

- L'algorithme donné teste en boucle si la valeur du compteur dépasse celle spécifiée par `FOOD_GENERATOR_DELAY` et la décrémente de `FOOD_GENERATOR_DELAY` si c'est le cas. Cela permet de gérer le cas où `dt` vaudrait plusieurs fois `FOOD_GENERATOR_DELAY`.

Par exemple, si `dt` était égal à trois fois `FOOD_GENERATOR_DELAY` et que l'on se contentait de mettre le compteur à zéro, on manquerait alors deux générations de nourriture. Ce test à l'aide d'une boucle permet de régler ce type de problèmes.

De plus, cela gère également le cas où `dt` aurait une valeur légèrement supérieure à `FOOD_GENERATOR_DELAY`, par exemple une fois et demi sa valeur. Deux appels successifs à `update()` vont alors faire générer trois instances de nourriture ; ce qui est le comportement attendu en moyenne.

- Nous vous recommandons de plus, pour des raisons d'efficacité, de ne faire appel qu'*une* seule fois par grandeur aux méthodes `Context.getConfig()` ; faites donc ces appels *avant* vos boucles et stockez le résultat dans une variable locale (non modifiable).
- Si vous souhaitez additionner ou soustraire un temps à un autre, utiliser les méthodes `Time plus(Time that)` ou `Time minus(Time that)` de la classe `Time`.

Remarquez que ces deux méthodes renvoient un `Time`. En effet, elles ne modifient pas l'objet sous-jacent mais en créent un nouveau. Par exemple, si vous souhaitez soustraite `t2` à `t1` il faut donc écrire `t1 = t1.minus(t2)`. Notez bien que c'est la *référence* `t1` que l'on modifie, et non pas le contenu qu'elle référence !

- Pour tester si un temps `t1` est supérieur ou égal à un temps `t2`, utilisez `t1.compareTo(t2) >= 0`.

Ouf !...

Classe `Environment` – implémentation

Nous allons enfin pouvoir mettre en place la première interaction entité-environnement comme annoncé dans le descriptif de cette étape. Nous allons donc pour cela entamer l'implémentation de la classe `Environnement`.

Pour commencer, la classe `Environment` doit hériter de l'interface `FoodGeneratorEnvironmentView` et doit être finale. Dotez là ensuite d'un attribut de type `FoodGenerator`, ainsi que d'une collection de nourritures, de type `List<Food>`. Ces deux attributs seront initialisés dans le constructeur par défaut. En ce qui concerne la liste de nourriture, instanciez-la avec une `LinkedList<Food>`.

En effet, certaines collections d'entités nécessiteront des suppressions en milieu d'itération. Par exemple, les instances de nourriture dont la quantité est nulle devront être retirées de la liste, car elles n'ont plus d'utilité au sein de l'environnement. De telles suppressions devront être efficaces. Les listes chaînées telles que `LinkedList` possèdent cette propriété désirée (ce n'est pas le cas des `ArrayList`).

Vous êtes désormais en mesure de pouvoir correctement définir la méthode abstraite `addFood()` déclarée dans l'interface `FoodGeneratorEnvironmentView`. Au cas où la `Food` passée en paramètre vaudrait `null`, cette méthode lancera une `IllegalArgumentException` (certaines méthodes de la classe `Utils` peuvent vous être utiles à cet égard !)

Définissez également la méthode

```
List<Double> getFoodQuantities()
```

qui retournera la liste des *quantités* associés à chaque nourriture présente dans l'environnement.

Terminons par la méthode `update(Time dt)` : pour le moment, elle se contente :

1. d'appeler la méthode

   ```
   void update(FoodGeneratorEnvironmentView, Time dt)
   ```

   de `FoodGenerator` ;

2. et de supprimer les nourritures qui sont épuisées (quantité inférieure ou égale à 0).

En fait, en l'état actuel, aucune entité ne consomme de nourriture et il n'y aura donc pas de suppression pour le moment, mais codez déjà cette partie qui vous sera utile par la suite.

La suppression d'éléments lors du parcours d'une collection se fait par le biais de la méthode `removeIf`, comme par exemple :

```
foods.removeIf(food -> food.getQuantity() <= 0);
```

Et voilà !

Il n'y a plus qu'à tester... (avant de soumettre !)

Tests et soumission

Tests locaux

Comme pour l'étape précédente, nous vous encourageons grandement à développer vos propres tests dans le fichier `ch/epfl/moocprog/tests/Main.java`.

Par exemple, pour cette étape-ci, vous pourriez *ajouter* (en plus de celles de l'étape précédente que nous vous conseillons de garder) des lignes de code comme :

```
Food f1 = new Food(tp2, 4.7);
Food f2 = new Food(tp3, 6.7);

System.out.println();
```

```
    System.out.println("Some tests for Food");
    System.out.println("Display : ");
    System.out.println(f1);
    System.out.println("Initial : " + f1.getQuantity()
                                    + ", taken : "
                                    + f1.takeQuantity(5.0)
                                    + ", left : " + f1.getQuantity());
    System.out.println("Initial : " + f2.getQuantity()
                                    + ", taken : "
                                    + f2.takeQuantity(2.0)
                                    + ", left : "
                                    + f2.getQuantity());

    final Time foodGenDelta = getConfig().getTime(FOOD_GENERATOR_DELAY);
    Environment env = new Environment();
    env.addFood(f1);
    env.addFood(f2);

    System.out.println();
    System.out.println("Some tests for Environment");
    System.out.println("Inital food quantities : " + env.getFoodQuantities());

    env.update(foodGenDelta);
    System.out.println("After update : " + env.getFoodQuantities());
```

qui donnerait les résultats suivants (en plus de ceux de l'étape précédente) :

```
Some tests for Food
Initial : 4.7, taken : 4.7, left : 0.0
Initial : 6.7, taken : 2.0, left : 4.7

Some tests for Environment
Inital food quantities : [0.0, 4.7]
After update : [4.7, 14.01472431185578]
```

Libre à vous, bien sûr, de les compléter par vos propres tests !

Soumission

Pour soumettre votre devoir au correcteur automatique, il faut créer un fichier ZIP contenant **TOUS VOS** fichiers sources, depuis la racine `ch`, ceux de cette étape mais aussi ceux de l'étape précédente. Concrètement, pour ce devoir ci, votre fichier ZIP doit donc contenir exactement les fichiers suivants :

```
ch/epfl/moocprog/Environment.java
ch/epfl/moocprog/Food.java
ch/epfl/moocprog/FoodGenerator.java
```

`ch/epfl/moocprog/FoodGeneratorEnvironmentView.java`
`ch/epfl/moocprog/Positionable.java`
`ch/epfl/moocprog/ToricPosition.java`

Note : si c'est plus pratique pour vous, vous *pouvez* aussi faire un zip qui contient tout le sous-répertoire `ch/epfl/moocprog`, y compris, donc, les fichiers que nous vous avons fournis. Ces fichiers seront simplement ignorés par le correcteur automatique (et remplacés par les nôtres). L'énoncé « *Procédure de soumission* » de la semaine 1 vous indique comment procéder. **Avant de passer à l'étape suivante, n'oubliez pas de faire une sauvegarde de l'étape en cours**, comme indiqué dans le même document.